**Node structure of binary tree :-**

|  |  |  |
| --- | --- | --- |
| Left | Data | Right |

Node structure :-

struct Node

{

struct \*left;

int data;

struct \*right;

};

Edges

|  |  |  |
| --- | --- | --- |
| 1000 | | |
| 2000 | 10 | 3000 |

20 is the parent of 40,50. Level – 1,Height = 2

Root, Parent of 20 &30 Level – 0, Height = 1



|  |  |  |
| --- | --- | --- |
| 2000 | | |
| 4000 | 20 | 5000 |

|  |  |  |
| --- | --- | --- |
| 3000 | | |
| 6000 | 30 | NULL |

30 is the parent of 60 Level – 1,Height = 2



|  |  |  |
| --- | --- | --- |
| 4000 | | |
| NULL | 40 | NULL |



|  |  |  |
| --- | --- | --- |
| 5000 | | |
| NULL | 50 | NULL |

|  |  |  |
| --- | --- | --- |
| 6000 | | |
| NULL | 60 | NULL |

40,50,60 are leaves, Level – 2, Height = 3

For the above Binary Tree :-

Nodes = 6 , Edges = 5 [ For n nodes, (n-1) edges are there ]

Here 40,50,60 are called leaves sinces these nodes have child count as 0.

40,50 are sibiing

Height = level + 1

Maximum nodes at that level 2 power of l (level) [ pow(2,l) ]

Height = 3 Total maximum nodes of above tree is 23 – 1 = 8 – 1 = 7

**Different ways to traverse through binary tree : -**

BFS = Breadth First Search

DFS = Depth First Search

1. **Level Order :-**

It is same as order of the given tree

Ex :- 10 20 30 40 50 60

1. **Inorder :-**

It means Left Subtree - Root - Right Subtree

|  |
| --- |
|  |
|  |
| 10 |
| Root |

Ex :- 40 20 50 10 60 30 70

|  |  |  |
| --- | --- | --- |
| Left | Root | Right |
| 40 | 20 | 50 |
| Left Subtree | | |

|  |  |  |
| --- | --- | --- |
| Left | Root | Right |
| 60 | 30 | 70 |
| Right Subtree | | |

1. **Post Order :-**

It means Left Subtree - Right Subtree - Root

Ex :- 40 50 20 60 70 30 10

|  |  |  |
| --- | --- | --- |
| Left | Right | Root |
| 40 | 50 | 20 |
| Left Subtree | | |

|  |  |  |  |
| --- | --- | --- | --- |
| Left | Right | Root |  |
| 60 | 70 | 30 |  | 10 |
| Right Subtree | | |  | Root |

1. **Pre Order :-**

It means Root - Left Subtree - Right Subtree

Ex :- 10 20 40 50 30 60 70

|  |
| --- |
| 10 |
| Root |

|  |  |  |
| --- | --- | --- |
| Root | Left | Right |
| 20 | 40 | 50 |
| Left Subtree | | |

|  |  |  |
| --- | --- | --- |
| Root | Left | Right |
| 30 | 60 | 70 |
| Right Subtree | | |

**Accessing nodes of binary tree using array :-**

Root = Null

Left = 2\*i+1

Right = 2\*i+2

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 |

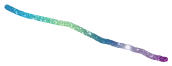
|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Inorder | Left of 10 | | | | | ROOT | Right of 10 | | |
| Left of 20 | | |  | Right of 20 | Left of 30 |  | Right of 30 |
| L (40) |  | R (40) |
| 80 | 40 | 90 | 20 | 50 | 10 | 60 | 30 | 70 |

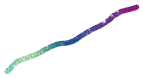


|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Postorder | Left of 10 | | | | | Right of 10 | | | ROOT |
| Left of 20 | | | Right of 20 |  | Left of 30 | Right of 30 |  |
| L (40) | R (40) |  |
| 80 | 90 | 40 | 50 | 20 | 60 | 70 | 30 | 10 |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Preorder | ROOT | Left of 10 | | | | | Right of 10 | | |
|  | Left of 20 | | | Right of 20 |  | Left of 30 | Right of 30 |
|  | L (40) | R (40) |
| 10 | 20 | 40 | 80 | 90 | 50 | 30 | 60 | 70 |

Edges







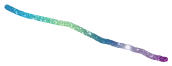


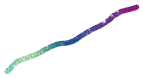




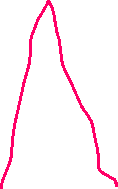
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Inorder | Left of 50 | | | | | | | ROOT | Right of 50 | | | | |
| Left of 40 | | |  | Right of 40 | | | Left of 80 | | |  | Right of 80 |
| L (30) |  | R (30) | L (45) |  | R (45) | L (70) |  | R (70) |
| 25 | 30 | 35 | 40 | 42 | 45 | 48 | 50 | 65 | 70 | 75 | 80 | 90 |

Edges

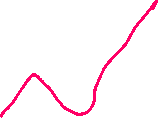
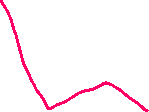
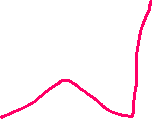












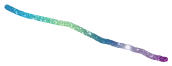




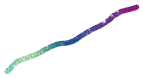
Direction of inorder from 25 to 90

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Postorder | Left of 50 | | | | | | | Right of 50 | | | | | ROOT |
| Left of 40 | | | Right of 40 | | |  | Left of 80 | | | Right of 80 |  |
| L (30) | R (30) |  | L (45) | R (45) |  | L (70) | R (70) |  |
| 25 | 35 | 30 | 42 | 48 | 45 | 40 | 65 | 75 | 70 | 90 | 80 | 50 |

Edges



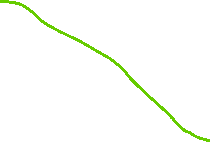
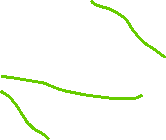








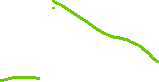






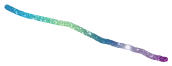


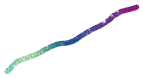
Direction of postorder from 25 to 50

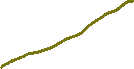


|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Preorder | ROOT | Left of 50 | | | | | | |  | | | | |
|  | Left of 40 | | | Right of 40 | | |  | Left of 80 | | | Right of 80 |
|  | L (30) | R (30) |  | L (45) | R (45) |  | L (70) | R(70) |
| 50 | 40 | 30 | 25 | 35 | 45 | 42 | 48 | 80 | 70 | 65 | 75 | 90 |

Edges



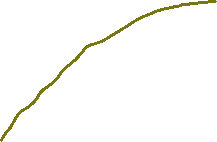




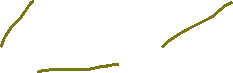












Direction of preorder from 50 to 90



**Construction of binary tree using Queue data structure :-**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 |

**Construction of binary tree using Linked Lists :-**

**Same Logic** Root = Null Left = 2\*i+1 Right = 2\*i+2

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1000 | |  | 2000 | |  | 3000 | |  | 4000 | |  | 5000 | |  | 6000 | |
| 10 | 2000 |  | 20 | 3000 |  | 30 | 4000 |  | 40 | 5000 |  | 50 | 6000 |  | 60 | NULL |
| Head | |  |  | |  |  | |  |  | |  |  | |  |  | |

|  |  |  |
| --- | --- | --- |
| root | | |
| 2000 | 10 | 3000 |

|  |  |  |
| --- | --- | --- |
| 3000 | | |
| 6000 | 30 | NULL |

|  |  |  |
| --- | --- | --- |
| 2000 | | |
| 4000 | 20 | 5000 |



|  |  |  |
| --- | --- | --- |
| 4000 | | |
| NULL | 40 | NULL |

|  |  |  |
| --- | --- | --- |
| 6000 | | |
| NULL | 60 | NULL |

****

|  |  |  |
| --- | --- | --- |
| 5000 | | |
| NULL | 50 | NULL |

****

temp = root temp->val > temp = temp->left temp->val < temp = temp->right

**Binary Search Tree :-**

**Rules**

Root

Root

Right

Left

1.Left subtree values are less than root 2. Right subtree values are greater than root

**Operaions on Binary Search Tree:-** Insert

Delete

Search

Inorder

Preorder

Postorder

Levelorder

Values :- 50 40 30 42 100 120 60 10 20 55 110 200 25



|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Inorder | 10 | 20 | 25 | 30 | 40 | 42 | 50 | 55 | 60 | 100 | 110 | 120 | 200 |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Postorder | 25 | 20 | 10 | 30 | 42 | 40 | 55 | 60 | 110 | 200 | 120 | 100 | 50 |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Preorder | 50 | 40 | 30 | 10 | 20 | 25 | 42 | 100 | 60 | 55 | 120 | 110 | 200 |

**Search operation :-**



Ex :-

Search(val) temp = root while true if temp->val == key :- print element found if temp->val > key :- temp = temp->left if temp->val < key :- temp = temp->right if temp == NULL :- element not found

**Search(20)**

1. Temp = root = 50

**50 == 20 50 < 20 50 > 20**

[ Since 3rd condition is correct we have to check left of temp i.e; **temp = temp->left** ]

1. Temp = 50->left = 40

**40 == 20 40 < 20** **40 > 20**

[ Since 3rd condition is correct we have to check left of temp i.e; **temp = temp->left** ]

1. Temp = 40->left = 30

**30 == 20 30 < 20** **30 > 20**

[ Since 3rd condition is correct we have to check left of temp i.e; **temp = temp->left** ]

1. Temp = 30->left = NULL

Since **temp == NULL**, we no need to check any of the three conditions and can say **element is not found** . So, 20 is not in the above binary tree.

**Search(55)**

1. Temp = root = 50

**50 == 55 50 < 55 50 > 55**

[ Since 2nd condition is correct we have to check right of temp i.e; **temp = temp->right** ]

1. Temp = 50->right = 60

**60 == 55 60 < 55** **60 > 55**

[ Since 3rd condition is correct we have to check left of temp i.e; **temp = temp->left** ]

1. Temp = 60->left = 55

**55 == 55 55 < 55** **55 > 55**

Since 1st condition is correct, we no need to check any of the three conditions and can say **element is found** . So, 55 is found in the above binary tree.

**Insert Operation :-**

Insert(val) temp = root while true if root == NULL :- root = NN(key) break if temp->val == key :- print element is already in tree break if temp->val > key :- if temp->left != NULL :- temp = temp->left else:- temp->left = NN(i.e; key) break if temp->val < key :- if temp->right != NULL :- temp = temp->right else:- temp->right = NN(i.e; key) break



**Insert(20)**

1. Temp = root = 50

**50 == 20 50 < 20 50 > 20**

[ Since 3rd condition is correct we have to check left of temp i.e; **temp = temp->left** **= 50->left = 40** **!= NULL**]

1. Temp = 50->left = 40

**40 == 20 40 < 20** **40 > 20**

[ Since 3rd condition is correct we have to check left of temp i.e; **temp = temp->left** **= 40->left = 30** **!= NULL**]

1. Temp = 40->left = 30

**30 == 20 30 < 20** **30 > 20**

[ Since 3rd condition is correct we have to check left of temp i.e; **temp = temp->left** **= 30->left = NULL**]

1. Temp = 30->left = NULL

Since **temp == NULL and 20 < 30** , we no need to check any of the three conditions and can **insert element at left of 30 as new node** .So, 20 is inserted as a left child of 30 in the above binary tree.









**Insert(55)**

1. Temp = root = 50

**50 == 55 50 < 55 50 > 55**

[ Since 2nd condition is correct we have to check right of temp i.e; **temp = temp->right = 50->right = 60 != NULL**]

1. Temp = 50->right = 60

**60 == 55 60 < 55** **60 > 55**

[ Since 3rd condition is correct we have to check left of temp i.e; **temp = temp->left = 60->left = 55 != NULL** ]

1. Temp = 60->left = 55

**55 == 55 55 < 55** **55 > 55**

Since 1st condition is correct, we no need to check remaining conditions and can say **element is already present in the binary search tree**.

**Insert(100)**

1. Temp = root = 50

**50 == 100 50 < 100 50 > 100**

[ Since 2nd condition is correct we have to check right of temp i.e; **temp = temp->right = 50->right = 60 != NULL**]

1. Temp = 50->right = 60

**60 == 100 60 < 100** **60 > 100**

[ Since 2nd condition is correct we have to check right of temp i.e; **temp = temp->right = 60->right = 65 != NULL** ]

1. Temp = 60->right = 65

**65 == 100 65 < 100** **65 > 100**

[ Since 2nd condition is correct we have to check right of temp i.e; **temp = temp->left** **= 65->right = NULL**]

1. Temp = 65->right = NULL

Since **temp ==NULL and 100 > 65**, we no need to check any of the three conditions and can **insert element at right of 65 as new node** .So, 100 is inserted as a left child of 65 in the above binary tree.







**Delete Operation :-**

Let’s work on the following example to understand the delete operation.

**50 40 80 30 45 70 90 25 35 42 48 65 100**









**There are three cases to delete an element**

**General procedure in deleting any of the node**

If root == NULL we have to return **NULL**  **temp = root parent = NULL** while(temp && temp->data != key) if temp->data > key :- **parent = temp ; temp =temp->left**  else :- **parent = temp; temp = temp->right**

1. Node has zero child 🡪 25, 35, 42, 48, 65, 100
2. Node has one child 🡪 70, 90
3. Node has two child 🡪 30, 45, 40, 80, 50

We want to return the res to main function in any case if res == NULL :- It prints **element not found** Else it prints **res free it’s memory**

**delete(25) :-**

For deleting Leave nodes ( temp->right == NULL && temp->left == NULL)

**res = temp** if parent->right != NULL && parent ->right->data== key :- **parent->right = NULL** else if parent->left != NULL && parent ->left->data== key:- **parent->left == NULL return res**

1. temp = root = 50 and parent = NULL

**50 == 25 50 < 25 50 >25**

[ Since 3rd condition is correct we have to check left of temp i.e; **parent = temp = 50 and temp = temp->left = 50->left = 40 != NULL** ]

1. temp = 40 and parent = 50

**40 == 25 40 < 25 40 >25**

[ Since 3rd condition is correct we have to check left of temp i.e; **parent = temp = 40 and temp = temp->left = 40->left = 30 != NULL ]**

1. temp = 30 and parent = 40

**30 == 25 30 < 25 30 >25**

[ Since 3rd condition is correct we have to check left of temp i.e; **parent = temp = 30 and temp = temp->left = 30->left = 25 != NULL** ]

1. temp = 25 and parent = 30

**25 == 25 25 < 25 25 >25**

[ Since 3rd condition is correct we have to check left of temp i.e; **parent = temp = 30 and temp = temp->left = 30->left = 25 != NULL** ]

Here 1st condition is correct, we no need to check remaining conditions and can copy 25 to res. Since 30->left = **25 != NULL** Now we have to make **30->left as NULL** by returning **res** to main the **element is printed on compiler and it’s memory is freed in the binary search tree**.









**delete(70) :-**

1. temp = root = 50 and parent = NULL

**50 == 70 50 < 70 50 > 70**

For deleting single child left node (temp->right == NULL)

**res = temp** if parent->right != NULL && parent ->right->data== key :- **parent->right = temp->left** else if parent->left != NULL && parent ->left->data== key:- **parent->left == temp->left return res**

[ Since 2nd condition is correct we have to check right of temp i.e; **parent = temp = 50 and temp = temp->right = 50->right = 80 != NULL** ]

1. temp = 80 and parent = 50

**80 == 70 80 < 70 80 > 70**

[ Since 3rd condition is correct we have to check left of temp i.e; **parent = temp = 80 and temp = temp->left = 80->left = 70 != NULL ]**

1. temp = 70 and parent = 80

**70 == 70 70 < 70 70 > 70**

[ Since 1st condition we no need to check remaining conditions and can copy 70 to temp. ]

Since 80->left = **70 != NULL and 70 == Key.** Therefore, **80->left = 70->left = 65** by returning **res = 70** to main the **element is printed on compiler and it’s memory is freed in the binary search tree**









**delete(90) :-**

For deleting single child right node (temp->left == NULL)

**res = temp** if parent->right != NULL && parent ->right->data== key :- **parent->right = temp->right** else if parent->left != NULL && parent ->left->data== key:- **parent->left == temp->right return res**

1. temp = root = 50 and parent = NULL

**50 == 90 50 < 90 50 > 90**

[ Since 2nd condition is correct we have to check right of temp i.e; **parent = temp = 50 and temp = temp->right = 50->right = 80 != NULL** ]

1. temp = 80 and parent = 50

**80 == 90 80 < 90 80 > 90**

[ Since 2nd condition is correct we have to check right of temp i.e; **parent = temp = 80 and temp = temp->right = 80->right = 90 != NULL ]**

1. temp = 90 and parent = 80

**90 == 90 90 < 90 90 > 90**

[ Since 1st condition we no need to check remaining conditions and can copy 90 to temp. ]

Since 80->right = **90 != NULL and 90 == Key.** Therefore, **80->right = 90->right = 100** by returning **res = 90** to main the **element is printed on compiler and it’s memory is freed in the binary search tree.**









**Delete (50) :-**

1. temp = root = 50 and parent = NULL

**50 == 50 50 < 50 50 > 50**

[ Since 1st condition we no need to check remaining conditions and can copy 70 to temp. ]

Since 50->right = **80 != NULL** and50->left = **40 != NULL and 50 == Key.** In this case,  **we cannot directly return the res = 50**  to main the **element is printed on compiler and it’s memory is freed in the binary search tree.**

**Before returning we to swap the 50 with it’s inorder predecessor or inorder successor**

**If we swap 50 with it’s inorder predecessor the tree will be like this**









**If we swap 50 with it’s inorder successor the tree will be like this**









|  |  |  |
| --- | --- | --- |
| **For deleting two child nodes**  **( temp->right != NULL && temp->left != NULL)** | | |
| In this case, swapping will occur, For this we have to declare two more pointer variables p and t.  And val as integer variable instead of key  **t = temp->right**  **p = NULL** while (t->left) :- **p = t**  **t = t->left** | if p != NULL :- **res = t val = t->data t->data = temp-> data temp->data = val p->left = t->right** return **res** | else :-  **res = t val = t->data t->data = temp-> data temp->data = val temp->right = t->right** return **res** |

**AVL TREE :-**

AVL Tree is a Self balancing tree.

**Balancing Factor :-**

It is the difference between maximum depth of left and maximum right of a node. i.e; (max depth of left – max depth of right)

It must be 0,-1,1

**Rotations :-**

1. Left Rotation
2. Right Rotation

We use the above rotations to balance the above tree.

**4 cases :-**

1. Right right case 🡪 Left rotation
2. Left left case 🡪 Right rotation
3. Right left case 🡪 Right rotation and Left rotation
4. Left right case 🡪 Left rotation and Right rotation

**Left left case** 🡪 **Right rotation**

**Right rotation :-**









**Example :-**

Let’s consider three numbers 15, 12 , 10 in place of X, Y, T1

0

2



0

0

1



0





**Right right case 🡪 Left rotation**

**Left rotation :-**









**Example :-**

0

Let’s consider three numbers 10, 12 , 15 in place of X, Y, T3

2



0

0

1





0

**Left right case 🡪 Left rotation and Right rotation**

**Example :-**

Let’s consider three numbers 15, 5 , 10 in place of X, Y, T2

2

2



-1

1









0

0



0

0

0





**Right left case 🡪 Right rotation and Left rotation**

**Example :-**

Let’s consider three numbers 15, 20 , 17 in place of X, Y, T2

2

2



1

1







0



0

0

0

0





Let’s work on this example to clearly understand the above all cases **15, 18, 12, 8, 54, 5, 14, 13, 9, 59, 20, 17, 21**

1

2



1

1



0





0

Here balancing factor of 12 is 2 which is unbalanced and this is the left left case. So we have to apply right rotation on 12,8,5 .

2

-2



1

0



2

0



1



0

Here balancing factor of 12 is 2 which is unbalanced and this is the right left case. So we have to apply right rotation on 12,14,13 and left rotation on 12,13,14 . Balanced factor of 15 also get balanced when it’s left subtree gets balanced.

2

-2



1

0



1

0



1

0



0

Here balancing factor of 8 is -2 which is unbalanced and this is right left case. To balance this first we have to apply right rotation on 12, 13, 14. Then, we have to apply left rotation on 8,12,9. Balanced factor of 15 also get balanced when it’s left subtree gets balanced.

0

0



2

0



1

1

0



0

0

0

Here balancing factor of 18 is 2 which is unbalanced and this is right right case. To balance this we have to apply left rotation on 18, 54, 59.

1

0



2

0



-1

0

-1

0



1

0

0



0

0

Here balancing factor of 54 is 2. we have to balance 54th left which is unbalanced and this is left right case. To balance this we have to apply left rotation on 18, 20, 21. Then, we have to apply right rotation on 54, 20,21.

0

0



0

0



-1

0

-1

0

0



0

0

0

0